Índice

1. Introducción

1.1. Motivación del proyecto

En 1971, Stephen Cook propuso en su trabajo[1] una nueva categoría de complejidad de problemas de decisión computacionales, a la que llamó problemas *NP-completos*. La caracterización de esta categoría se hace sobre estas dos propiedades:

- Todos los problemas *NP-completos* pueden ser verificados en tiempo $O(p(n))$, donde $p(n)$ es un polinomio en función de n el tamaño de la instancia del problema.
- Todos los problemas en *NP* pueden ser reducidos en tiempo $O(p(n))$ a algún problema *NP-completo*, donde $p(n)$ es un polinomio en función de n el tamaño de la instancia del problema que es reducido.

Ahora bien, fueron Cook y Leonid Levin quienes encontraron, de forma independiente, el primer problema en esta categoría *NP-completos*: el problema de la *satisfacción booleana (SAT)*. Un año después, Richard Karp identificó otros 21 problemas en esta categoría [2], los cuales tenían la notoria característica de que para ellos no se conoce un algoritmo polinomial (en función del tamaño de la instancia) que les de solución, una cualidad que comparten todos los problemas en esta clase, junto al hecho de que todos estos problemas ocurren con una marcada frecuencia en el área de la computación. Sin embargo, la característica más especial de éstos es el segundo ítem de arriba: encontrar un algoritmo polinomial para tan sólo uno de ellos es encontrar un algoritmo polinomial para todos.

De modo pues que la motivación para este proyecto estriba en el hecho de que SAT fue el primer problema que se demostró que pertenece a *NP-Completos* y que todos los problemas en esta clase son reducibles en tiempo polinomial a él. Siendo así y bajo el supuesto de que estas reducciones a SAT se caractericen por polinomios de bajo grado y coeficientes pequeños, cualquier mejora en tiempo que se pueda realizar a los algoritmos exponenciales

hoy conocidos para resolver el problema *SAT* es una mejora para los algoritmos exponenciales conocidos para los demás problemas en *NP-completos*.

1.2. Breve descripción del problema

Llamaremos *cláusula* a la disjunción de un conjunto finito de variables booleanas, cada una de las cuales puede ocurrir con polaridad positiva (no negada: x_i) o con polaridad negativa (negada $\overline{x_i}$). Ejemplos de cláusulas son: $(x_1 \wedge x_2)$, $(\overline{x_3})$, $(\overline{x_1} \wedge x_3)$.

Una *fórmula* es en cambio la conjunción de un conjunto finito de cláusulas. Por ejemplo: $F_1 : \; (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_3})$ es una cláusula.

El *problema de la satisfacción booleana (SAT)* consiste de la forma general de las instancias al problema y de la pregunta:

1. La forma general de las instancias: Dados un conjunto finito de variables booleanas $x_1, x_2, \ldots, x_n$ y una fórmula booleana $F(x_1, x_2, \ldots, x_n)$ en forma normal conjuntiva (CNF).

2. La pregunta cuya respuesta se quiere determinar: ¿existe una asignación de valores de verdad a las variables $x_1, \ldots, x_n$ tal que la fórmula sea verdad?

2. Diseño

2.1. Descripción general del algoritmo empleado

El programa propuesto sigue el esquema general del algoritmo DPLL que presentaron Martin Davis, Hilary Putnam, George Logemann y Donald Loveland para decidir el problema de satisfacción booleana.

Seguidamente presentamos el esquema general del algoritmo DPLL, que hemos adaptado de [4]:

```
  status = preprocess();
if (status != UNKNOWN) return status;
while(true) {
```

```
  // Fase de seleccion de variable a asignar.
  decide_next_branch();
  while (true) {
    // Fase de deduccion.
    status = deduce();
    if (status == CONFLICT) {
      conflict_result = analyze_conflict();
      if (conflict_result == 0)
        return UNSATISFIABLE;
      else backtrack;
    }
    else if (status == SATISFIABLE)
      return SATISFIABLE;
    else break;
  }
}
```

Uno de los puntos en que esta presentación del algoritmo diverge del que presenta [4] es en el análisis de conflicto y en la posibilidad de efectuar un *backtracking* no cronológico. En el programa que aquí describimos, no se realiza sino una versión muy simplificada de análisis de conflicto, en donde no hay aprendizaje de cláusulas y el *backtracking* se hace sólo de un nivel de decisión en un nivel de decisión (cronológico).

Entre las razones por las que se escogió el algorimo DPLL están:

1. Este algoritmo es la base para las implementaciones en los *SAT-solvers* más eficientes conocidos.

2. El esquema general del algoritmo es bastante sencillo de implementar.

2.2. Fase de selección de variables a asignar

2.3. Fase de deducción

Una vez que se ha seleccionado una variable para asignar junto al valor que se le asignará, se inicia la fase de deducción del algoritmo DPLL. Es en esta fase que se procede a la identificación y propagación de cláusulas

unitarias. En esta sección describiremos los algoritmos de propagación de restricciones booleanas y de identificación y eliminación de literales puros.

2.3.1. Identificación y propagación de cláusulas unitarias

Tal como se indica en [4] y [**?**], con el estudio de implementaciones del algoritmo DPLL que se han realizado con el pasar de los años, sugiere que la propagación de cláusulas unitarias como mecanismo de deducción parece ser el más eficiente que se ha encontrado hasta ahora.

La *propagación de cláusulas unitarias* consiste en ubicar cúales cláusulas de la fórmula —dada ésta en forma normal conjuntiva— están compuestas de un sólo literal. Estas cláusulas son llamadas *cláusulas unitarias* y se satisfacen con asignar a éste único literal el valor de verdad correspondiente. Lo explicamos con un ejemplo: En la fórmula $F_1 :\ (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_3})$, la única cláusula unitaria es $\overline{x_3}$. Si se asigna $x_3 := 0$, queda la nueva fórmula $F_2 :\ (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2})$ y bajo el supuesto de que $x_3 = 0$ se tiene que F_2 es satisfactible si y sólo si F_1 lo es.

Para la implementación de este mecanismo de deducción, se escogió la implementación de los *2-watched literals* descrita en las fuentes [3], [4] y [**?**] y que fue propuesta con el programa *zChaff*.

Las razones para escoger esta implementación de *2-watched literals* para la identificación de cláusulas unitarias son las siguientes:

1. En pruebas de ejecución[4] se ha observado que el comportamiento de la implementación por *2-watched literals* requiere de menor tiempo que implementaciones como la de *SATO* —implementación a la que [4] se refiere como *Head/Tail lists*— y considerablemente menor tiempo que la implementación por contadores.

2. La implementación de *2-watched literals* no requiere que se realicen operaciones sobre los *watchers*, o sobre el conjunto de datos que permite directamente determinar cuáles son las cláusulas unitarias, cuando se realiza el *backtracking* a un nivel de decisión anterior. Las implementaciones de *SATO* y contadores sí requieren estas operaciones. Esta ventaja es señalada por el trabajo [3].

Entre las desventajas que trae consigo la implementación de *2-watched literlas* está:

1. Cuando es necesario mover alguno de los *watchers*, porque la variable a la que apuntan en la cláusula resulta asignada, se debe buscar una nueva variable no asignada—puede que ni exista tal variable— en la misma cláusula. Entonces, en el peor caso, para identificar una cláusula unitaria, la implementación de *2-watched literals* tendrá que *recorrer todos los literales de una misma cláusula* en busca de esta variable no asignada.

2.3.2. Eliminación de literales puros

La *eliminación de literales puros* en una fórmula dada en forma normal conjuntiva consiste en ubicar primero cuáles son las variables booleanas que sólo ocurren con una polaridad en la fórmula. Ahora bien, estos literales que ocurren con una única polaridad en toda la fórmula no condicionan la satisfacción de la fórmula; es decir, si se eliminaran todos los literales puros, la fórmula resultante es satisfactible si y sólo si asignando a los literales puros los valores de verdad que los satisfagan, se logra que la fórmula original lo sea. Lo explicamos con un ejemplo: En la fórmula siguiente $F_1 : \; (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_3})$ el único literal puro es x_1, de forma que F_1 será satisfactible si y sólo si asignando a x_1 el valor de verdad se logra que $F_2 : \; (x_2 \vee \overline{x_3}) \wedge (\overline{x_2}) \wedge (\overline{x_3})$ sea satisfecha.

Para la implementación de este mecanismo de deducción se hizo un recorrido por todas las cláusulas de la fórmula anotando cuál es la polaridad que se ha observado para cada literal. Si en un momento del paseo por las cláusulas de la fórmula se encuentra con un literal que ocurre con una polaridad distinta a la ya observada anteriormente para ese literal, se descarta que esa variable sea un literal puro. Para más detalles consultar más adelante.

3. Detalles de implementación

La implementación de la propagación de cláusulas unitarias con 2 testigos por cláusula ([4] y [**?**] los llama *2-watched literals*) empleada por *zChaff* asocia

a cada literal x_i, $i \in 1, \ldots, n$ un par de listas, la primera de ellas tiene como elementos a todas las cláusulas en las que la el literal x_i ocurre no negado (polaridad positiva) como testigo o *watched literal*. La segunda lista asociada a x_i tiene como elementos a todas las cláusulas en las que el literal $\overline{x_i}$ ocurre como testigo o *watched literal*.

El programa que proponemos está orientado por los cambios que se efectúan, durante la ejecución, sobre una variable global de nombre `sat_st` que es la única con el tipo `SAT_status` en todo el programa. Este tipo de dato registra:

1. La información que es necesario preservar de la instancia del problema de satisfacción que se ha leído y que se pretende resolver.
2. El estatus de resolución de un problema de satisfacción en cualquier momento dado.

Por esta razón podríamos afirmar que este es el tipo de dato más importante de todo el programa. Presentamos ahora la definición del tipo `SAT_status`

```
typedef struct SAT_status{
    int num_vars;
    int num_clauses;
    clause *formula;
    list *pos_watched_list;
    list *neg_watched_list;
    stack backtracking_status;
    int *model;
} SAT_status;
```

, para comentar sus campos con detalle:

- El atributo `formula`, representa la fórmula en forma normal conjuntiva. Se trata de un arreglo de cláusulas, cada una de tipo `clause`.
- En la implementación que aquí se describe, se optó por por los campos `pos_watched_list` y `neg_watched_list` en el tipo `SAT_status`. Cada uno de éstos es un arreglo de cabezas de listas, de forma que `pos_watched_list[i]` sea la cabeza de la lista cuyos elementos son las cláusulas en las que el literal x_i ocurre como *watcher*. Análogamente ocurre con `neg_watched_list[i]`: es la cabeza de la lista cuyos elementos son las cláusulas en las que el literal $\overline{x_i}$ ocurre como *watcher*.

- El campo `model` del tipo `SAT_status` es un arreglo de enteros tal que `model[i]` es el valor de asignación que se prueba para la variable x_i. El `model` indica cuál nodo de la arborescencia del *backtracking* se está considerando en un determinado instante de la ejecución[1].

- Se incluye el campo `num_clauses` en el tipo `SAT_status`, para poder recorrer el arreglo `formula` de todas las cláusulas que componen la fórmula.

A continuación comentaremos algunos detalles sobre la variable global `sat_st`. La razón por la que se escogió a `sat_st` de tipo `SAT_status` como variable global, en lugar de pasarla como parámetro entre las sucesivas llamadas a funciones durante la ejecución del algoritmo son:

1. El pasaje del parámetro `sat_st` a cada una de las funciones supone un costo acumulado muy grande a lo largo de la ejecución de todo el programa. Cuando bien pudiera ahorrarse la operación de empilar esa parámetro en cada llamada.

2. Si se pasara una referencia a `sat_st` como parámetro a cada función, se incurriría en un costo adicional, en comparación con la alternativa de tener a `sat_st` como variable global, por la indirección que es necesario ejecutar en cada subrutina por cada vez que se quiera acceder a los campos de esta variable.

3.1. Descripción de la implementación de las cláusulas

Para la implementación de cada cláusula se definió el siguiente tipo de dato `clause`:

```
typedef struct clause{
    int size;
    variable* head_watcher;
    variable* tail_watcher;
    variable* literals;
} clause;
```

[1]Véase la sección 3.2.1 que describe la arborescencia implícita que se recorre en el *backtracking*.

, que a continuación describiremos campo por campo.

1. Los apuntadores `head_watcher` y `tail_watcher` señalan cuáles son los literales testigos —*watched literals*— de una cláusula. En virtud de que en la fase de propagación de restricciones booleanas se optó por implementar la propagación de cláusulas unitarias con los *2-watched literals* según se describe en [4], cada cláusula exige dos apuntadores a variables en la misma cláusula.

2. Como cada cláusula es una disjunción de literales, optamos por representarla como un arreglo de variables llamado `literals`. Para poder recorrerlo es necesario almacenar su tamaño, que estará almacenado en el campo `size` de la cláusula.

3.1.1. Ventajas de la implementación escogida para las cláusulas

La implementación de los literales que componen una cláusula en un arreglo de variables apuntado por `literals` implica:

- Una rapidez de acceso en tiempo constante a cada literal de la cláusula. Hecho que resulta de particular utilidad en los recorridos a través de los literales de cada cláusula que son efectuados durante la detección de literales puros y durante la actualización de los *watchers* o testigos que permiten identificar cláusulas unitarias.

  Recuerde el lector que se ha mencionado en la sección **??** que una de las desventajas de las implementación por *2-watched literals* es que en el peor caso hay que recorrer todos los literales de una cláusula para determinar si ésta es unitaria o no.

- Un ahorro de espacio para los apuntadores, el cual sería necesario si la conjunción de literales en las cláusulas se implementara con una lista enlazada.

3.2. Descripción de la implementación del *Backtracking*

3.2.1. Árbol implícito del *Backtracking*

Toda implementación de *Backtracking* es un recorrido *Depth-First Search* sobre una arborescencia implícita. Esta descripción implícita de la arborescencia a recorrer exige que se defina cuáles son sus nodos y para cada nodo, cuáles son sus nodos sucesores. En el caso que nos concierne, los nodos son de la forma:

$$[x_{i_1} = \mathbb{B}, x_{i_2} = \mathbb{B}, \ldots, x_{i_k} = \mathbb{B}],$$

donde $0 \leq k \leq n$, con n el número de variables de la instancia del problema de satisfacción a resolver y $x_{i_k} = \mathbb{B}$ indica que la variable booleana x_{i_k} tiene un valor booleano (sea 1 ó 0) asignado. Imponemos adicionalmente una condición a los nodos de esta arborescencia y es que la asignación hecha a las variables del nodo: $x_{i_1}, \ldots, x_{i_k}$ *no haga que la fómula no se pueda satisfacer*. Las x_{i_k} denotan variables booleanas distintas $\forall k \in \{1, \ldots, n\}$.

Ahora, dado un nodo $[x_{i_1} = \mathbb{B}, x_{i_2} = \mathbb{B}, \ldots, x_{i_k} = \mathbb{B}]$ sus sucesores son todos los nodos de la forma: $[x_{i_1} = \mathbb{B}, x_{i_2} = \mathbb{B}, \ldots, x_{i_k} = \mathbb{B}, x_{i_{k+1}} = \mathbb{B}]$.

El *backtracking* implementado busca encontrar en la arborescencia —en caso de que exista— un nodo de la forma

$$[x_1 = \mathbb{B}, x_2 = \mathbb{B}, \ldots, x_n = \mathbb{B}],$$

y que se corresponde con una asignación de valores de verdad a todas las variables que hace que la fórmula dada sea satisfecha —si n es el número total de variables booleanas.

3.2.2. Estructuras de datos que apoyan la implementación del *bactracking*

El *backtracking* se implementó iterativo en lugar de recursivo, por los motivos que se señalan en la sección 3.2.3. Para ello fue necesario trabajar explícitamente con una pila de elementos de un nuevo tipo de dato llamado `decision_level_data`. Este tipo almacena la información que caracterizan a cada nodo del árbol implícito que se recorre en el *backtracking*[2] y que deben

[2] En [4] les llaman niveles de decisión.

ser conservados en caso de que el algoritmo se encuentre con un nodo parcial que no tiene sucesores; esto es, con un nodo $[x_{i_1} = \mathbb{B}, x_{i_2} = \mathbb{B}, \ldots, x_{i_k} = \mathbb{B}]$, $k < n$ tal que la asignación de cualquier otra variable no logra satisfacer la fórmula.

Presentamos entonces el tipo `decision_level_data`:

```
typedef struct decision_level_data{
    variable assigned_literal;
    int missing_branch;
    list propagated_var;
} decision_level_data;
```

que a continuación describimos:

1. El campo `assigned_literal` contiene el valor y el nombre del literal asignado en un determinado nivel de decisión. Expresado de otra forma, si $[x_{i_1} = \mathbb{B}, x_{i_2} = \mathbb{B}, \ldots, x_{i_k} = \mathbb{B}]$, $k < n$, $v_k \in \mathbb{B}$ es un nodo de la arborescencia recorrida con el *backtracking*, en el momento en que se consideran una nueva variable $x_{i_{k+1}}$ con un valor booleano se ha creado un nuevo nivel de decisión caracterizado por un elemento del tipo `decision_level_data` en el programa. Este elemento tendrá en su campo `assigned_literal` a la variable $x_{i_{k+1}}$ con su valor.

2. El campo `missing_branch` es un valor booleano que es cierto si y sólo si se ha explorado la asignación de `assigned_literal` con un sólo valor de verdad. Es decir, si $[x_{i_1} = \mathbb{B}, x_{i_2} = \mathbb{B}, \ldots, x_{i_k} = v_k]$, $k < n$, $v_k \in \mathbb{B}$ es un nodo de la arborescencia recorrida con el *backtracking*, en la que la variable x_{i_k} fue la última variable asignada con un valor booleano determinado, `missing_branch` será cierto si y sólo si el conjunto de asignaciones $[x_{i_1} = \mathbb{B}, x_{i_2} = \mathbb{B}, \ldots, x_{i_k} = \overline{v_k}]$, $k < n$ *todavía no ha sido estudiado* si es nodo o no.

3. El campo `propagated_var` es la cabeza de una lista cuyos elementos son todas las variables booleanas

3.2.3. Ventajas de la implementación escogida para el *backtracking*

Entre las ventajas de la implementación iterativa para el *backtracking* está:

- Resultará más sencillo modificar el programa para implementar un *bactracking* no cronológico, que si se hubiera implementado el *backtracking* de manera recursiva.

3.2.4. Desventajas de la implementación escogida para el *backtracking*

Quizás la única desventaja de la implementación iterativa respecto a la recursiva para el *backtracking* sea la mayor dificultad que supone el manejo explícito de la pila, que en el caso recursivo se maneja implícitamente con la pila de llamadas a subrutinas.

4. Dificultades encontradas

5. Instrucciones de operación

Para emplear la aplicación, escribir en la consola el comando

```
# kec_o_sat_s -f inputfilename -o outputfilename
```

donde:

- `inputfilename` es el nombre del archivo que contiene la instancia del problema SAT a resolver. Esta instancia debe estar en el formato DIMACS. (Consultar http://logic.pdmi.ras.ru/ basolver/dimacs.html para información sobre este formato.)

- `outputfilename` es el nombre del archivo que contendrá los resultados generados tras correr el algoritmo.

Para más detalles acerca de la operación, leer el archivo `README` que se incluye con la distribución del programa.

6. Estado Actual

El programa se encuentra totalmente operativo.

7. Conclusiones y recomendaciones

Referencias

[1] Cook, Stephen: "The complexity of theorem-proving procedures". *ACM*, 1971.

[2] Karp, Richard: "Reducibility Among Combinatorial Problems". *Complexity of Computer Computations*. 1972. . *ACM*, 1971.

[3] Lynce, I. y Marques-Silva, J.: "Efficient Data Structures for Fast SAT Solvers". Reporte Técnico. *Cadence European Laboratories, Instituto de Engenharia de Sistemas e Computadores*. 2001.

[4] Zhang, Lintao y Malik, Sharad: "The Quest for Efficient Boolean Satisfiability Solvers".

[5] Zhang, Lintao: "Searching for truth: Techniques for satisfiability of boolean formulas". *Tesis de doctorado. Princeton University*. 2003.